ER Catcher: A Static Analysis Framework for Accurate and Scalable Event-Race Detection in Android

Navid Salehnamadi, Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek
School of Information and Computer Sciences
University of California, Irvine, USA
{nsalehna,aalshayb,iftekha,malek}@uci.edu

ABSTRACT

Android platform provisions a number of sophisticated concurrency mechanisms for the development of apps. The concurrency mechanisms, while powerful, are quite difficult to properly master by mobile developers. In fact, prior studies have shown concurrency issues, such as event-race defects, to be prevalent among real-world Android apps. In this paper, we propose a flow-, context-, and thread-sensitive static analysis framework, called ER Catcher, for detection of event-race defects in Android apps. ER Catcher introduces a new type of summary function aimed at modeling the concurrent behavior of methods in both Android apps and libraries. In addition, it leverages a novel, statically constructed *Vector Clock* for rapid analysis of happens-before relations. Altogether, these design choices enable ER Catcher to not only detect event-race defects with a substantially higher degree of accuracy, but also in a fraction of time compared to the existing state-of-the-art technique.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Program Analysis, Android, Concurrency, Event-Race Detection

1 INTRODUCTION

Modern mobile frameworks promote the development of highly concurrent software applications, or "apps" for short. In the case of Android, concurrency is ingrained in all facets of app behavior: (1) an app's components run within their own threads, (2) components can simultaneously interact with components within and outside of the app by exchanging Intent messages, (3) the components at any point in time may receive lifecycle (e.g., `onStart()` and `onPause()`) and system (e.g., location change, battery low) callbacks without any guarantees as to the order in which they may occur. To aid the developers with development of concurrent software, Android provisions several new concurrency constructs in the form of libraries, such as `AsyncTask` and `Looper`. Nevertheless, concurrency is a major source of confusion for developers [26] and remains among the top 5 reasons for defects in Android apps [51].

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416639

The conventional techniques for detection of concurrency issues in Java, such as data-race defects [15, 30, 34, 46], are not readily applicable for Android [10, 38]. They neither explicitly consider the event-driven model of app behavior, nor support the new concurrency constructs in Android. More recently, researchers have investigated both dynamic [10, 20, 29] and static [17, 22, 45] analysis techniques for detection of concurrency issues in Android. Dynamic analysis techniques proposed so far are limited in their capability, as they miss true event races due to their limited coverage of the app behavior [22]. Existing static analysis techniques [17, 22, 45] fail to accurately identify many event races due to (1) imprecise modeling of concurrency behavior in Android, and (2) adoption of analyses that are innately flow-, context-, thread-insensitive. Moreover, the existing static analysis techniques are slow, e.g., nAdroid [17] takes about 50 minutes on average to analyze an app. Besides that, these works support only a small and fixed set of concurrency libraries, e.g., SARD [45] only considers asynchronous invocations by `Handler`, `Activity`, and `Thread` APIs, and cannot be easily extended to support other libraries. Furthermore, the tools realizing these techniques are either unavailable ([22, 45]), or closed-source ([17]).

In this paper, we introduce ER Catcher, a static analysis framework for effective detection of event-race defects in Android apps that aims to overcome the shortcomings of prior works. Three novel concepts set our work apart from the prior techniques and allow us to succeed where others have failed.

First, ER Catcher relies on a new type of summary function, called *Concurrency-aware Summary Function (CSF)*, for modeling the concurrent behavior of methods in both Android apps and libraries. The *CSF* for each app method is automatically extracted, while the set of *CSF*s representing concurrent behavior of methods comprising a library are manually constructed as a one-time effort. ER Catcher processes methods in an app's call graph in a reverse topological order, thereby improving the performance of analysis, i.e., eliminating the need to reanalyze the same method multiple times. Furthermore, by modeling each library (e.g., `AsyncTask`, `Handler`) as a set of *CSF* specifications, the implementation of ER Catcher is completely separated from its support of Android libraries. This, in turn, allows one to add support for new or modified Android libraries by simply providing ER Catcher with the proper *CSF* specifications, and without requiring changes to the ER Catcher's implementation.

Second, ER Catcher builds its analysis on an abstraction representation of app, called *Context and Concurrency-aware Call Graph (C^3G)*. In C^3G, each method call is represented in terms of its three execution *states*: *invocation*, *start*, and *end*. The states are connected

through edges that distinguish between *synchronous* and *asynchronous* interactions. Unlike a conventional call graph, C^3G is aware of the running thread of each method. The fine-grained representation of app behavior in C^3G enables ER Catcher to perform *flow-, context-, and thread-sensitive* analysis. This increased sensitivity of analysis addresses the limitations imposed by imprecise modeling in prior techniques that affect their accuracy.

Third, ER Catcher employs a novel, statically computed data structure, called *Static Vector Clock (SVC)*, to efficiently analyze the happens-before relations in an app. SVC is inspired by the concept of Vector Clock [24] from the domain of distributed systems for determining the order of events in such systems. *SVC* enables ER Catcher to quickly query happens-before relations for a subset of events suspected to be involved in an event race, thereby making its analysis substantially faster than prior techniques.

Besides the above, ER Catcher is the first of its kind, completely open-source static analysis tool for event-race detection in Android [2]. Based on our extensive empirical evaluations, ER Catcher outperforms the state-of-the-art static event-race detector, nAdroid [17], in terms of accuracy (13% more precise) and performance (12 times faster). Moreover, our experiments show ER Catcher is both practical and scalable—capable of analyzing 90% of 500 randomly selected real-world apps from F-Droid repository [3] in under 5 minutes per app.

The remainder of this paper is organized as follows. Section 2 provides an example of event-race defect in Android that is used to describe the approach. Sections 3 describes the details of our approach, while Section 4 presents our experimental evaluation. Section 5 summarizes the related work and finally Section 6 concludes and discusses the future work. The tool, research artifacts, and proofs of soundness can be found on the companion website [2].

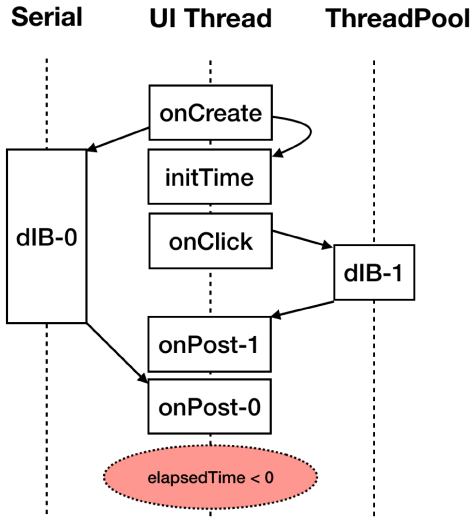
2 ILLUSTRATIVE EXAMPLE

Figure 1a shows the code snippet of a "screen time" tracking app that logs the time a user has spent on an app on an external server and displays it on the screen. `TrackTimeActivity` is created when the app starts. It immediately sends a request to the server to record the initial activation time. From then on, whenever the user clicks on `recordButton`, the app requests the server to record the time and displays the elapsed time on screen.

According to the code in Figure 1a, `onCreate` method asynchronously invokes `InitTime.run` (`initTime` in short) and `RecordTimeTask.doInBackground` (`dIB` in short) methods (lines 8-9). `initTime` method runs on the UI (main) thread, since it is initiated by a `Handler` of main `Looper`. `Looper` is a FIFO queue wrapper for a thread in Android, enabling tasks to be queued for execution by the thread. `Handler` is responsible for enqueueing tasks in the associated `Looper` using the `post` method. Here, the task is `initTime` method, which simply initializes `startTime`, `elapsedTime`, and `timeView` variables (lines 20-24). The `executeOnExecutor` method called on line 9 is an `AsyncTask` library API that eventually results in the invocation of `dIB` method in `SERIAL_EXECUTOR`, i.e., a construct that enqueues tasks and executes them sequentially in a single thread. `dIB` sends `currentTime` to the server through a blocking `send` method (line 29). Once `dIB` is finished (the data has been sent

```
1  class TrackTimeActivity extends Activity {
2   long startTime, elapsedTime;
3   TextView timeView;
4   Button recordButton;
5   void onCreate(){
6     recordButton = findViewById(...)
7     timeView = findViewById(...)
8     new Handler(getMainLooper()).post(new InitTime());
9     new RecordTimeTask().executeOnExecutor(SERIAL_EXECUTOR,
        System.currentTimeMillis());
10    recordButton.setOnClickListener(new OnClickListener(){
11     public void onClick(View view) {
12       Executor executor = THREAD_POOL_EXECUTOR
13       // executor = SERIAL_EXECUTOR;
14       new RecordTimeTask().executeOnExecutor(executor,
        System.currentTimeMillis());
15     }
16    });
17  }
18
19  class InitTime implements Runnable{
20   public void run() {
21     startTime = -1;
22     elapsedTime = 0;
23     timeView.setText("Initializing");
24   }
25  }
26
27  class RecordTimeTask extends AsyncTask {
28   Long doInBackground(Long currentTime){
29     send(currenttime);
30     return currentTime;
31   }
32   void onPostExecute(Long currentTime){
33     if (startTime < 0)
34       startTime = currentTime;
35     elapsedTime = currentTime - startTime;
36     timeView.setText(Long.toString(elapsedTime));
37   }
38  }
39 }
40
```

(a) The code of the app



(b) The timeline of the event race

Figure 1: An event race example in Android

to the server), `RecordTimeTask` invokes `onPostExecute` (`onPost` in short) on the UI thread and passes the `currentTime` to it. This method sets `startTime` (if has not been set so far), updates `elapsedTime`, and displays it on `timeView` widget (lines 32-37).

The UI event handler for `recordButton` is `onClick`, which executes `RecordTimeTask`. Point to note, `dIB` invoked through `onClick` (lines 11-15) runs on `THREAD_POOL_EXECUTOR`, i.e., a construct that collects tasks and executes them in parallel without any order.

To distinguish between the invocations of `RecordTimeTask` through `onCreate` and `onClick`, we add an index to them: `dIB-0` and `onPost-0` when invoked by `onCreate`, and `dIB-1` and `onPost-1` when invoked by `onClick`.

The logically correct behavior for this app should follow a specific execution order. Namely, `initTime` should happen before `onPost-0`, and `onPost-0` must happen before `onPost-1` in order to show the correct elapsed time on the screen. This assumption, however, can be violated, since there is no *happens-before* relation between `onPost-0` and `onPost-1`, as depicted in Figure 1b. Due to network communication latency, the execution time of `send` method (line 29) is non-deterministic. It is thus possible that `dIB-0` finishes after `dIB-1`, resulting in `onPost-1` method to execute prior to `onPost-0`. Consequently, `startTime` is set to the time that `recordButton` is clicked rather than the time `onCreate` is invoked, which means a negative value for time is shown on screen. This issue can be resolved by uncommenting line 13 to execute all `RecordTimeTasks` in a serial mode. That would enforce `onPost-1` to be executed after `onPost-0`.

A static analysis approach for effectively detecting event-race defects in Android needs to be flow-, context-, and thread-sensitive:

- **Flow-Sensitive:** a flow-insensitive approach would ignore the statement ordering, thereby reporting false positives. For example, the only reason `initTime` happens before `onPost-0` is that `initTime` is invoked before `dIB-0` in the body of `onCreate` (lines 8 and 9).
- **Context-Sensitive:** A context-insensitive approach does not distinguish between the invocation of two methods under different execution contexts, thereby misses true event races. In the above example, there is one `onPostExecute` method, but it can be executed under different contexts (through `onCreate` or `onClick`). Without considering the context of execution, the occurrence of such an event race could not be detected.
- **Thread-Sensitive:** A thread-insensitive approach either falsely assumes all tasks are executing in one thread, or conservatively assumes no tasks share threads. A thread-insensitive approach that assumes all tasks are running in the same thread would miss true event races shown in the example of Figure 1a. On the other hand, if both `dIB-0` and `dIB-1` are actually running in the same thread (in the fixed version of Figure 1a), a thread-insensitive approach that assumes no tasks share threads cannot detect the actual happens-before relation and reports a false positive.

Furthermore, a practical solution needs to be (1) **Scalable**, to make it suitable for execution during the development process, (2) **Extensible**, to accommodate the evolution of concurrency constructs in modern mobile platforms, and (3) **Available**, to allow practitioners to make use of it, and to enable researchers to build on top of it.

The aforementioned criteria constitute the requirements for ER Catcher, described next.

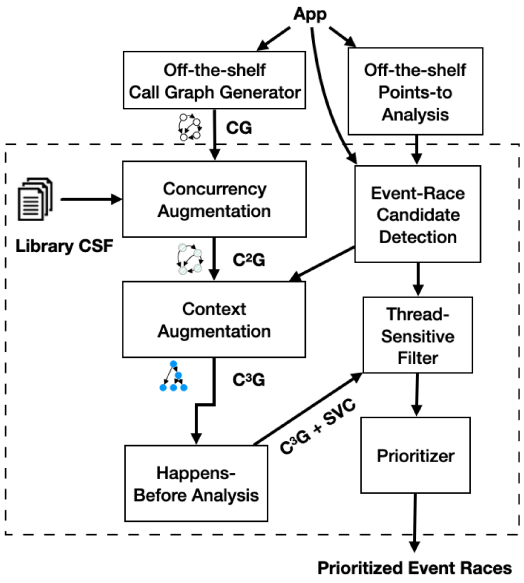


Figure 2: An overview of ER Catcher

3 APPROACH

Figure 2 shows the components of ER Catcher. We explain the details of each component in this section. For brevity, we elide the details of call-graph generation and points-to analysis, since we employ existing off-the-shelf solutions [7, 25].

3.1 Concurrency Augmentation

Consider the implementation of the illustrative app (Figure 1a), which overrides `doInBackground` and `onPostExecute` methods of `AsyncTask` library. Here, most of the semantics of concurrent behavior does not exist in the source code of the app itself. First of all, there is no explicit invocation of these methods. Moreover, the code itself does not reveal a particular order of execution. Besides that, these methods may run in different threads. However, from our understanding of the `AsyncTask` library, we know that by invoking `executeOnExecutor` method, `doInBackground` will be implicitly invoked on the thread which can be determined by the first parameter (e.g., `SERIAL_EXECUTOR`), followed by `onPostExecute` that runs on the `UI` thread. Although this information is embedded in the implementation of libraries, including the libraries in the analysis would make the approach slow and inefficient. More importantly, this information may not be extracted precisely due to the limitations of static analysis. To overcome this challenge, we introduce the notion of *Concurrency-aware Summary Function (CSF)*.

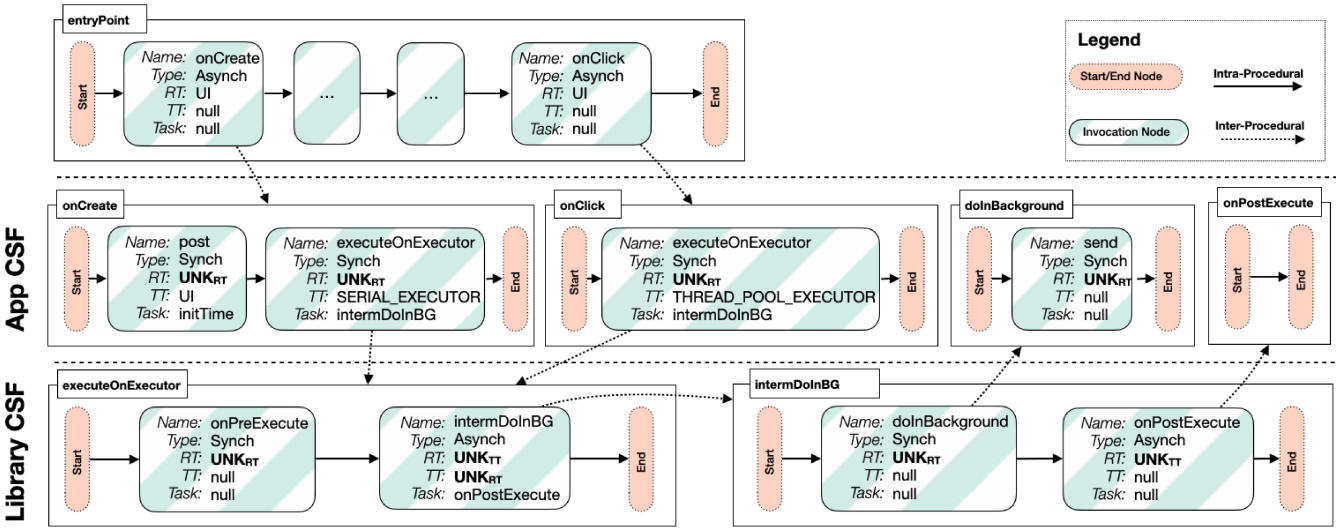


Figure 3: A subset of the C^2G of app in Figure 1a

The concept of *CSF* is inspired by *summary-based static analysis* [18, 40], where for each method a model representing the method's behavior is constructed and subsequently reused to expedite the analysis. We adapted this technique to model only concurrency-relevant information, e.g., the synchronous versus asynchronous nature of invocation calls, or the type of thread that an outgoing method call will execute on.

As shown in Figure 3, a *CSF* models a method call in terms of its three execution *states*: *invocation*, *start*, and *end*. We formally define *CSF* of a method m to be a Directed Acyclic Graph (DAG) represented as (N, E). Here, N consists of a start node and an end node indicating the initiation and termination of m respectively, and zero or more invocation nodes indicating the methods that are called as a result of execution of m. E is the set of directed edges such as (n_1, n_2), where n_1 dominates n_2, i.e., all paths starting from the entry of a method reaching n_2 has to pass through n_1. The edges of a *CSF* capture the intra-procedural flow that is missing in a conventional call graph.

An invocation node has five attributes ($Name$, $Type$, RT, $Task$, TT), where $Name$ is the name of the callee's method, $Type$ indicates the type of invocation (synchronous or asynchronous), RT is the running thread identifier that the callee method will run on, and $Task$ is the callback method that will eventually run on the task thread (TT) at some point in future.

We augment the call graph of an app, obtained using FlowDroid [7], with the concurrency behavior of its methods modeled in *CSF*s to arrive at a *Concurrency-aware Call Graph*, called C^2G. Figure 3 depicts a subset of the C^2G for the illustrative example (recall Figure 1a). Each large white box corresponds to a node in the call graph, while the internals of each box represent the *CSF*. The top row in Figure 3 shows the `entryPoint` method, which similar to prior work (FlowDroid [7]) is artificially created to emulate the lifecycle of an Activity.

The second row consists of the defined/overridden methods in the app. For instance, the *CSF* of `onCreate` consists of start, two invocation, and end nodes (due to limited space, some invocations, such as `findViewById`, are not shown). The first invocation node in `onCreate` represents line 8 in Figure 1a that invokes `post` (*Name*) method synchronously (*Type*) on `onCreate`'s thread (*RT*) and eventually `initTime` (*Task*) will be executed on `UI` thread (*TT*). Note that, certain contextual information, such as specific threads executing some of the methods, are unknown at this point, denoted in bold as *UNK*, e.g., UNK_{RT} in `post` invocation node is the running thread of `onCreate` that is not yet resolved.

The bottom row depicts the *CSF*s of library methods called by the app. The *CSF* of `executeOnExecutor` in `AsyncTask` library shows that this method first synchronously invokes `onPreExecute` on its own running thread, UNK_{RT}, then asynchronously invokes `intermDoInBG` method on another thread, UNK_{TT}. Moreover, the last invocation node passes UNK_{RT} as the task thread (TT), to model the fact that the result of computations performed in the background should eventually be returned to the main caller's thread. The other *CSF*, `intermDoInBG`, first synchronously invokes `doInBackground`, and next asynchronously invokes `onPostExecute` on UNK_{TT}. Note that, the actual implementation logic of `AsyncTask` library is much more complex than its corresponding summaries, e.g., it consists of several exception handlers and various classes, such as `WorkerRunnable`. However, the summaries are sufficiently modeling the concurrent behavior of the library for our analysis with a few *CSF* nodes.

To construct the C^2G, we expand each method in the call graph of an app with its corresponding *CSF* representation. The *CSF*s of concurrency library methods are provided as an external artifact, while the *CSF*s of app methods are automatically extracted. Each concurrency library method is associated with a *CSF* specification and a helper function that determines the attributes of its invocation nodes.

For each app method, the *CSF* nodes and edges are extracted using the control-flow graph of its body. Next, for each invocation node of a concurrency library method encountered in the app logic, the corresponding helper function determines the attributes such as *RT* and *TT*. Finally, to capture the order of calls more accurately, we change the source of call-graph edges to invocation nodes.

We model the concurrent behavior of Android components (i.e., `Activity`, `Service`, and `BroadcastReceiver`) using *CSF*s as well. An artificially created `entryPoint` method emulates the invocation of a component's callback methods according to the lifecycle of the corresponding component type. We automatically generate the *CSF* of `entryPoint` methods similar to app methods, except the callbacks are invoked asynchronously in main thread.

Currently, ER Catcher supports the following concurrency-related constructs in Android: `Thread`, `Looper`, `Handler`, `AsyncTask`, `IntentService`, `ServiceConnection`, and lifecycle methods of `Activity`, `Service`, and `BroadcastReceiver`. Note that ER Catcher analyzes both app code (written by the app developers) and app libraries, e.g., an advertisement library. However, if a library introduces its own concurrency mechanism, ER Catcher requires its corresponding *CSF* for precise analysis. For more details on the implementation of Library *CSF*, please visit the ER Catcher's website [2].

3.2 Event-Race Candidate Detection

In parallel to the construction of fine-grained models representing the concurrent behavior of an app, ER Catcher analyzes the app to identify a list of all candidate event races (recall Figure 2). The crude event races identified at this stage are then filtered in the subsequent steps. An event race is defined as a triplet $(Stmt_1, Stmt_2, F)$, where $Stmt_1$ and $Stmt_2$ are executing in methods M_1 and M_2, and F is a field (representing a memory location statically). These triplets have to satisfy the following properties: (1) $Stmt_1$ and $Stmt_2$ have access to memory location F, and at least one of the accesses is a write; and (2) there is no happens-before relation between $Stmt_1$ and $Stmt_2$

For example, the illustrative app of Figure 1a has two potential event races involving the memory location `elapsedTime`:
(`initTime`:22, `onPostExecute`:35, `elapsedTime`)
(`onPostExecute`:35, `onPostExecute`:35, `elapsedTime`)

In this component, we do not have any information about the context of methods; therefore, we use a conservative approach and presume no happens-before relation exists between the statements. We use an off-the-shelf points-to analysis technique (SPARK [25]) to determine the statements in two methods are accessing the same memory location.

3.3 Context Augmentation

For precise event-race detection, we need to account for the execution context (recall Section 2). To that end, we need to augment C^2G with contextual information to resolve the unknown entries such as UNK_{RT} and UNK_{TT} in *CSF*s. One naive approach for determining the contexts of all methods is to traverse the call graph along the edges, starting from the entry point. However, such an approach does not scale, because the number of potential paths is $\mathbf{O}(2^n)$, where n is the number of methods. To address this challenge, we use dynamic programming together with two filters (*Race Involvement* and *Synchronous Substitution*), which substantially prune the analysis and memory space. Once the contexts are determined, the remaining parts (running and task threads) can be determined by simply propagating threading information through the invocation caller sites. The result of this component is *Context- and Concurrency-aware Call Graph*, called C^3G.

Figure 4 depicts the process of determining the contexts for a subset of the illustrative example. For brevity, only the *CSF* nodes involved in the context augmentation are shown here. The edges annotated with *S* are synchronous call-graph edges.

We process the methods of C^2G, shown in Figure 4a, in a reverse topological order.[1] During this process, a method can be in the *visited*, *visiting*, or *non-visited* stages. For each visiting method m, we first apply *Race Involvement* filter: if neither m nor any of m's descendants are involved in an event race, we do not process m, e.g., method `findViewById` in Figure 4b is pruned. This filter reduces the analysis space drastically. The Event-Race Candidate Detection component, discussed earlier, provides us with a list of methods that may be involved in an event race.

If m passes this filter, for each incoming edge to its start node, we make a copy of m and all of its descendants to make its incoming edges unique. We then connect each edge to the start node of its corresponding copy (making the incoming edge of m's start node unique). For example, in Figure 4b, the start node of method `onPost` has only one incoming edge; therefore, there is only one copy of `onPost` in Figure 4c.

If the invocation of a method is synchronous, e.g., `onCreate` → `executeOnExecutor`, it does not impact the concurrent behavior of the app. In this situation, we apply *Synchronous Substitution* filter by (1) removing the callee method, (2) connecting the outgoing edges of caller method to the callee method's children, (3) annotating these edges with the caller to maintain the sequence of invocations. For example, Figure 4c shows *Synchronous Substitution* of `executeOnExecutor`. The method `executeOnExecutor` is eliminated and replaced with two annotated edges, representing alternative execution contexts. This filter substantially improves the utilized memory and processing time of subsequent steps. Note that since we are processing the methods in the app call graph in a reverse topological order, all descendants of m are already visited.

Figure 4d shows the final stage of context augmentation. As mentioned earlier, for each incoming edge of `intermDoInBG`'s start node (the edges from `onCreate` and `onClick`), a copy of `intermDoInBG` and its descendants (`onPost`) is created, making the incoming edge of each node unique.

Once the contexts of all methods are determined, the remaining two parts (running and task threads) can be determined by propagating the thread information starting from the entry point. The thread of the entry point is already known, since it runs on the UI thread and there is no task thread (as depicted in Figure 3). Using the C^3G edges we propagate these information. For example, in Figure 3, once the caller sites of `executeOnExecutor` are resolved to be `onCreate` and `onClick`, the specific thread

[1] If C^2G is cyclic, we apply 1-unrolling [19] procedure and eliminate cycles.

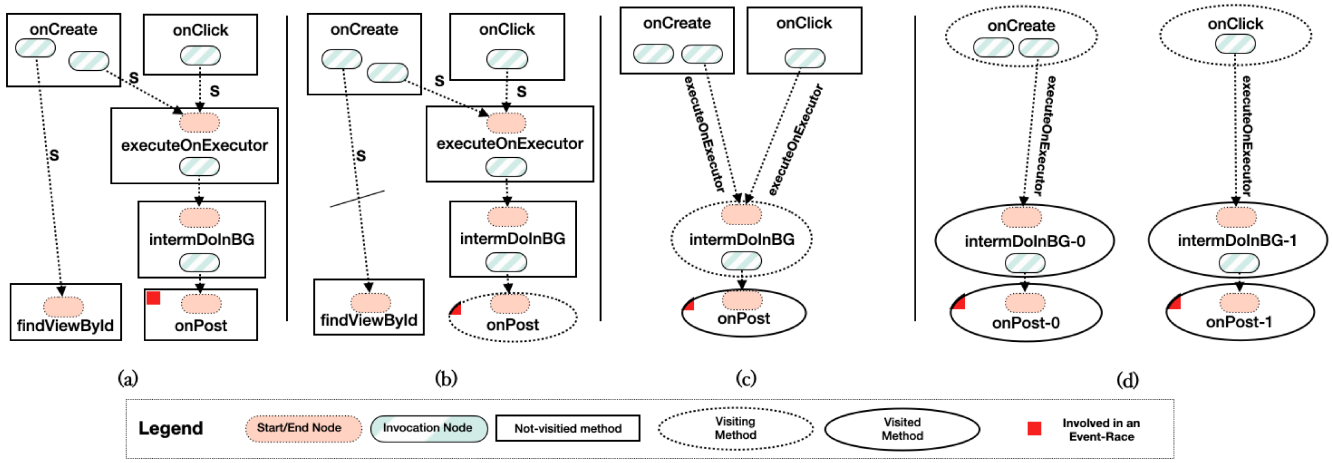


Figure 4: Process of Context Augmentation: a) subset of the C^2G of the app in Figure 1a, b) Race Involvement filter, c) Synchronous Substitution filter, and d) cloning contexts.

used to execute tasks in the background, as modeled by UNK_{RT} in `intermDoInBG`, can be determined. In the case of reaching `executeOnExecutor` through `onCreate`, UNK_{RT} of `intermDoInBG` is determined to be the `SERIAL_EXECUTOR`, while in the alternative case of reaching `executeOnExecutor` through `onClick`, UNK_{RT} of `intermDoInBG` is determined to be `THREAD_POOL_EXECUTOR`.

At this stage, all contextual information are determined. The final C^3G of the example in Figure 1a can be seen in Figure 5.

3.4 Happens-Before Analysis

For precisely determining event races, we need to determine the happens-before relation, denoted as $\prec$, among the execution states of methods (i.e., start, invocation, and end). To that end, we developed an efficient approach for statically reasoning about the happens-before relations in an event-driven program, called *Static Vector Clock* (*SVC*). Our solution is inspired by the notion of *Vector Clock* [24]—a well-known algorithm for determining partial ordering of events in a distributed system, whereby timestamps associated with messages exchanged among the distributed nodes are used to establish a logical clock. The conventional Vector Clock relies on the following basic principle to adjust the logical clock of a distributed system: the transmission of a message must precede its receipt in time. Similarly, in our work, we leverage the execution states of methods (nodes) represented in C^3G to construct a logical clock. We rely on three principles: **(principle 1)** for a given thread of execution, a method's invocation state must precede its start state, and a method's start state must precede its end state; **(principle 2)** a method's invocation state precedes another method's invocation state if the call statement of the former dominates that of the latter in the control-flow graph; and **(principle 3)** in a FIFO thread, a method's end state precedes the start state of another method, if the former invocation state precedes the latter invocation state. The proof of correctness for these principles can be found on the companion website [1].

Figure 5 illustrates how we compute the value of SVC for each node using the C^3G of the illustrative app. Each method call is represented in terms of three nodes with suffixes `.i` (invocation state), `.s` (start state), and `.e` (end state). Moreover, each node has a numerical identifier, e.g., `onCreate.i` is the invocation state of `onCreate` identified as node 1. The nodes are divided into three zones representing the threads that they are running on. For the sake of clarity, the main entry point method that invokes `onCreate` and `onClick` methods is not shown here.

The value of *SVC* for each node is shown above it as a vector. Formally, in an event-based system with $|T|$ threads, the SVC value of a node n is a vector $SVC(n) =< S_1, \cdots, S_{|T|} >$, where S_i is the *minimized* set of nodes occurring in thread T_i before node n. S_i is minimized when there does not exist any happens-before relation between any of its members. For example, the *SVC* value of node 20, meaning the last nodes occurring before node 20 in Serial, UI Thread, and ThreadPool are 5, 19, and 13, respectively. Here, since *SVC* is minimized, node 14 is not in $SVC(20)$, even though it happens before node 20, i.e., the invocation state of `onPost` occurs before its end state. Because $14 \prec 19 \prec 20$, the inclusion of 19 in the *SVC* is sufficient. On the other hand, $SVC(9)_2$ contains two nodes ($\{3, 6\}$), since there is no happens-before relation between the invocation state of `onClick` and the end state of `onCreate`.

Given the *SVC* of an app, we can quickly determine the existence of happens-before relation among its nodes:

Theorem 1. *Node a_1 happens before node a_n, $a_1 \prec a_n$, if there exists a sequence of nodes $(a_1, \cdots, a_n)$ such that $\forall 1 \leq i < n\ a_i \in SVC(a_{i+1})$.*

Theorem 2. *Two nodes a and b may happen without any order if there exists two threads, t_1 and t_2, such that $SVC(a)_{t_1} \prec SVC(b)_{t_1}$ and $SVC(b)_{t_2} \prec SVC(a)_{t_2}$*

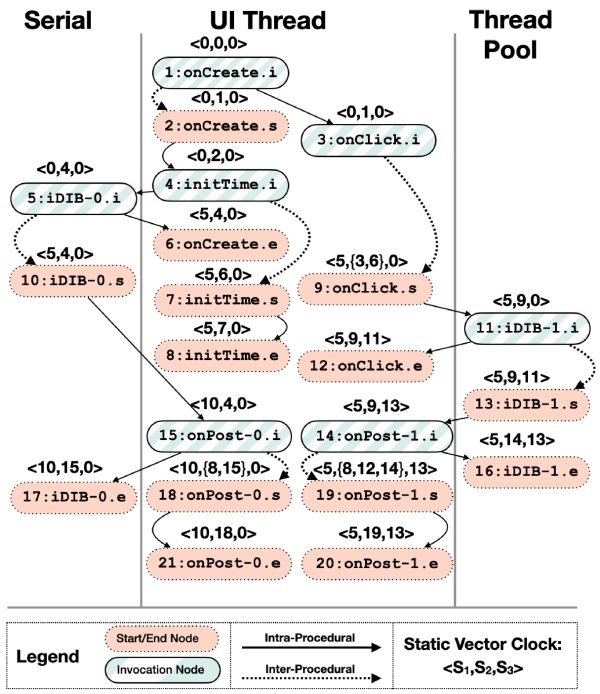


Figure 5: The C^3G of the illustrative app in Figure 1a annotated with Static Vector Clock values.

Algorithm 1 *SVC* calculation

Require: N (C^3G nodes), E (C^3G edges), T (set of threads)

1: **for each** $n \in N$ **do**
2: $\quad SVC^0(n) \leftarrow < \emptyset_0, \cdots, \emptyset_{|T|} >$
3: **for each** $n_1 \rightarrow n_2 \in E$ **do**
4: $\quad t \leftarrow n_1.t$
5: $\quad SVC^0(n_2)_t \leftarrow SVC^0(n_2)_t \cup n_1$
6: $i \leftarrow 0$
7: **repeat**
8: $\quad i \leftarrow i + 1$
9: $\quad SVC^i(n) \leftarrow SVC^{i-1}(n)$
10: $\quad \mathbb{O} \leftarrow$ the topological order of graph $(N, Inv(SVC^i))$
11: $\quad \mathcal{HBM} \leftarrow$ an empty dictionary from nodes to methods
12: $\quad$ **for each** $n \in \mathbb{O}$ **do**
13: $\quad\quad$ **if** $n = m.i \wedge n.t$ is FIFO **then**
14: $\quad\quad\quad$ **for each** $m' \in \mathcal{HBM}[n]_t$ **do**
15: $\quad\quad\quad\quad SVC^i(m.s)_t \leftarrow SVC^i(m.s)_t \cup m'.e$
16: $\quad\quad\quad$ Minimize $SVC^i(m.s)_t$
17: $\quad\quad\quad \mathcal{HBM}[n]_t \leftarrow m$
18: $\quad\quad$ **for each** $n' \in Inv(SVC^i)(n)_t$ **do**
19: $\quad\quad\quad \mathcal{HBM}[n'] \leftarrow \mathcal{HBM}[n'] \cup \mathcal{HBM}[n]$
20: **until** $SVC^i = SVC^{i-1}$
21: $SVC \leftarrow SVC^i$

The proof of correctness for these theorems can be found on the companion website [1]. We can determine if there is a happens-before relation between two nodes using Theorem 1. For example, $7 \prec 21$ because there exists a sequence of nodes, $(7, 8, 18, 21)$, where each element belongs to the SVC of its next element, or $7 \in SVC(8)$, $8 \in SVC(18)$ and $18 \in SVC(21)$. Using Theorem 2 we can determine if there is no happens-before relation between two nodes. For example, there is no happens-before relation between nodes 20 and 21, since we have $SVC(20)_1 = 5 \prec SVC(21)_1 = 10$ and $SVC(21)_3 = 0 \prec SVC(20)_3 = 13$.

Algorithm 1 shows how SVC is computed. We first initialize the SVC using C^3G edges as shown on lines $1-5$ ($n.t$ indicates the thread that node n is running on). This initialization satisfies the first two principles: call-graph edges enforce a method's invocation state happens before its start state (principle 1), while *CSF* edges enforce the happens-before relations implied by the control-flow graph (principle 2). The rest of algorithm satisfies the third principle using a fixed-point iteration method, i.e., update SVCs according to *Same Task-Queue Order* until the values of SVCs do not change.

Same Task-Queue Order ensures if two methods, m_1 and m_2, run on the same FIFO thread and $m_1.i$ happens before $m_2.i$ then m_1 finishes before m_2 (or $m_1.e \prec m_2.s$). For example, because the invocation states of methods `onClick` and `initTime` (nodes 3 and 4) happen before the invocation of `onPost-1` (node 14), we have $\{8, 12\} \subset SVC(19)_2$ (the end states of `onClick` and `initTime` happen before the start state of `onPost-1`). Note that even though `initTime` and `onClick` happen before `onPost-1`, there is no happens-before relation between their end states and `onPost-1` invocation state.

A naive approach to apply *Same Task-Queue Order* is to query happens-before relation between all pairs of method invocation states and update SVCs accordingly. However, we can avoid the query cost by traversing an ordered DAG $\big(N, Inv(SVC)\big)$, where the nodes are N and the directed edges are the set of (n_1, n_2) such that $n_1 \in SVC(n_2)$. One property of this ordered DAG, denoted by $\mathbb{O}$ for short, is that if $n_1 \prec n_2$ then n_1 precedes n_2 in $\mathbb{O}$. For example, the node identifiers in Figure 5 show the position of nodes in $\mathbb{O}$ and there is no $n_1 \prec n_2$ where the id of n_1 is greater than the id of n_2. We store the set of methods where their invocations occur before node n in $\mathcal{HBM}[n]$ while iterating over $\mathbb{O}$, since all the nodes that happen before n are already visited; as a result, there is no need to query happens-before for each pair of the invocation nodes.

Algorithm 1 initializes $\mathbb{O}$ and $\mathcal{HBM}$ on lines $10-11$, then iterates over $\mathbb{O}$. For each n running in a FIFO thread that is the invocation state of method m, the algorithm appends the end node of methods in $\mathcal{HBM}[n]_t$ to the SVC of m's start node (lines $13-15$). Since $SVC^i(m.s)_t$ may be updated, it should be minimized (line 16). To increase the performance of the algorithm, $\mathcal{HBM}[n]_t$ is set to m, since n is the latest invoked node in thread t for the remaining nodes (line 17). Finally, the algorithm propagates $\mathcal{HBM}[n]$ to the nodes in $Inv(SVC^i)(n)_t$ which is the set of nodes like n' that $n \in SVC^i(n')_t$ (lines $18-19$). This process repeats until SVC^i does not change.

3.5 Thread-Sensitive Filter

In this component, we use C^3G and SVC to prune the false event races produced by Event-Race Candidate Detection component (recall Figure 2). Given an event race, we check if there is a happens-before relation between the corresponding

methods. Method m_1 happens before method m_2 if all end states of m_1 happen before start states of m_2. We remove all event races in which there is a happens-before relation between their method calls. For example, we filter out the event race (`initTime`:22, `onPostExecute`:35, `elapsedTime`), because as depicted in Figure 5, there is a happens-before relation between all end states of `initTime` and start states of `onPost`: `initTime.e` $\prec$ `onPost-0.s` and `initTime.e` $\prec$ `onPost-1.s`. However, for the event race (`onPostExecute`:35, `onPostExecute`:35, `elapsedTime`), there is neither a happens-before relation between `onPost-0.s` and `onPost-1.e`, nor between `onPost-0.e` and `onPost-1.s`. We thus report it as a possible event race.

To further prune the false event races, two filters (*If-Guard* and *Null-At-End*) are designed for Use-after-Free (UF) defects. A UF defect is a harmful event race where one memory access makes a memory location free (writes `null`) and another access uses (reads) it, resulting in a `NullPointerException` to be thrown. *If-Guard* removes the UF defects where the read access is guarded by a null-checking condition, e.g., in `if (f!= null) f.use();` the field `f` will not be accessed if it is `null`. In addition, *Null-At-End* filters UF defects where the memory location is not reassigned with a value other than `null` after becoming free e,g,. in `f = null; f = new F();` the field `f` is not null.

In addition to the above-mentioned filters that are sound, we also provide several heuristics that are unsound, but in practice can significantly reduce the false warnings, as described next.

3.6 Prioritization

The reported event races by the previous component include all possible event races that need to be reviewed by developers. To facilitate this manual process, we prioritize the detected event races by our confidence in their existence. Our confidence is inversely related to the degree of over-approximation in the static analysis, i.e., less over-approximation in detecting an event race leads to more confidence about its existence. Due to the over-approximation of static analysis and our conservative approach, we do not filter event races for which we have incomplete information.

We prioritize event races according to the number of satisfied over-approximation properties. **(Reachability)** We prioritize event races that are reachable, i.e., there are paths in the call graph from the entry-point to both methods of a reachable event race. The thread-sensitive filters are not applied on unreachable event races because their thread information is unknown. **(Must-Alias)** Event races that involve statements accessing *must-alias* fields have priority over *may-alias* fields. Two fields are must-alias if they always point to the same memory location. **(Known-Thread)** Event races where the threads of their methods are known have more priority than others. For example, when we are not certain about the corresponding `Looper` of a `Handler` (which can be the main thread `Looper` or a custom thread `Looper`), we assume its messages are handled in an unknown thread that dispatches messages in parallel.

4 EVALUATION

This section discusses our experimental evaluation to answer the following research questions:

- **RQ1** How accurate is ER Catcher in detecting true event races?
- **RQ2** How fast does ER Catcher analyze real-world apps?
- **RQ3** How effective is ER Catcher in filtering false event races?
- **RQ4** What is the impact of modeling concurrency in improving the overall accuracy of static analysis?

We evaluate ER Catcher using three different datasets. First, we use a set of benchmark apps containing event-race defects, called BenchERoid [39]. For these apps the ground truth is known, allowing us to report the precision, recall, and F1 score of ER Catcher. Second, we use a "Curated" dataset of 31 real-world apps with event-race defects that have been confirmed, through either dynamic analysis, or code commit messages. We collected these apps by reviewing the prior literature [17, 21, 29] and crawling the open-source repositories. Finally, we evaluate the scalability and effectiveness of ER Catcher using 500 randomly selected apps from F-Droid [3]. Table 1 summarizes the datasets used for evaluation.

We compare ER Catcher with nAdroid [17], the state-of-the-art static *Use-after-Free (UF) event-race* detector that is available publicly, but not entirely open-source. Since nAdroid detects only UF event races, to make the comparison fair between ER Catcher and nAdroid, we configure ER Catcher to report only UF event races. We also tried to empirically compare ER Catcher to several other tools, namely SIERRA [22], SARD [45], ASYNCCLOCK [20], and EventRacer [10]. Since none of these tools are available, we contacted the corresponding authors. Unfortunately, despite multiple attempts, the authors either did not respond to us or confirmed their inability to release their tool. We provide a qualitative discussion of the differences between ER Catcher and these other techniques in Section 5.

Table 1: Properties of datasets used in our experiments.

Dataset	# Apps	Criteria	Average	Median	Min	Max
BenchERoid dataset	34	#Methods	27	25.5	18	51
		#Components	1	1	1	3
		Size (KB)	1123	1042	931	1646
Curated dataset	31	#Methods	1563	1066	78	5155
		#Components	19	12	1	138
		Size (KB)	3850	2069	29	26445
F-Droid dataset	500	#Methods	1041	499	10	6432
		#Components	9	5	1	544
		Size (KB)	4171	2000	10	97676

4.1 RQ1: Accuracy

We ran ER Catcher and nAdroid on the 34 apps provided by BenchERoid to measure their respective accuracy. The results are shown in Table 2. The actual number of event races in the benchmark apps are shown in column 2. nAdroid uses two different filters (sound and unsound) for reducing false event races; we report both of them in columns 3 and 4, respectively. We report the event races identified by ER Catcher in column 5 and the prioritized event races (recall Section 3.6) in column 6. We categorize the apps provided by

BenchERoid into four groups. The first group is non-UF event races. We do not consider them in comparing the accuracy of ER Catcher with nAdroid, since nAdroid does not report non-UF event races. The second and third groups consist of apps containing UF event races related to flow- and thread-sensitivity, respectively. The rest of the apps are placed in the "Other UF" category.

Table 2: Benchmark Result. The general event races (highlighted rows) were not considered for the accuracy metrics. ⊛True Positive ◯False Negative *False Positive.

App Name	# Ground Truth Event Races	nAdroid Sound	nAdroid Unsound	ER Catcher	ER Catcher Prioritized
Non-UF Event Races					
SingleActivity7	1	◯	◯	⊛*	⊛
SingleActivity8	2	◯◯	◯◯	⊛⊛**	⊛⊛*
Service5	1	◯	◯	⊛*	⊛
AsyncTask5	1	◯	◯	⊛*	⊛
AsyncTask6	0			*	
Flow-Sensitive UF					
SingleActivity2	0	*	*		
SingleActivity5	2	⊛⊛**	⊛⊛**	⊛⊛	⊛⊛
SingleActivity6	2	⊛⊛*	⊛⊛*	⊛⊛	⊛⊛
AsyncTask2	0	*			
Looper1	0				
Thread-Sensitive UF					
SingleActivity3	1	◯	◯	⊛	⊛
AsyncTask1	1	⊛	◯	⊛	⊛
AsyncTask3	1	⊛	◯	⊛	⊛
Thread1	1	⊛*	◯	⊛*	⊛*
Thread2	1	⊛*	◯	⊛*	⊛*
Looper2	1	⊛	⊛	⊛	⊛
Service3	1	◯	◯	⊛	⊛
Service4	0			*	*
Other UF					
MultiComp1	2	⊛⊛	⊛⊛	⊛⊛	⊛⊛
Receiver	1	⊛	⊛	⊛	⊛
Service1	0	*	*	*	*
Service2	1	⊛◯◯	⊛◯◯	⊛⊛◯	⊛⊛◯
LifeCycle1	3	⊛⊛⊛	⊛⊛⊛	⊛⊛⊛	⊛⊛⊛
LifeCycle2	0				
LifeCycle3	2	⊛⊛	⊛⊛	⊛⊛	⊛⊛
LifeCycle4	2	⊛⊛	⊛⊛	⊛⊛	⊛⊛
SingleActivity1	1	⊛	⊛	⊛*	⊛
SingleActivity4	0	***	***	**	**
AsyncTask4	1	⊛	◯	⊛	⊛
Executor1	1	⊛	⊛	⊛	⊛
Executor2	1	⊛*	◯	◯	◯
TimerTask1	1	⊛	⊛	⊛	⊛
TimerTask2	1	⊛	⊛	⊛	⊛
Looper3	1	⊛	⊛	⊛	⊛
Total ⊛ (higher is better)		27	21	**29**	**29**
Total ◯ (lower is better)		4	10	**2**	**2**
Total * (lower is better)		13	9	7	**6**
Precision		%67	%70	%80	**%82**
Recall		%87	%67	**%93**	**%93**
F1		%75	%68	%86	**%88**

Overall, ER Catcher is significantly more accurate than nAdroid. ER Catcher achieves 80% precision and 93% recall, compared to nAdroid's 67% precision and 87% recall. Due to its flow-sensitivity, ER Catcher is able to filter all of the false positives reported by nAdroid under the Flow-Sensitive UF category. Due to its thread-sensitivity, ER Catcher is able to identify additional true event races compared to nAdroid under the Thread-Sensitive UF category.

Although ER Catcher is more accurate than nAdroid, it has its own limitations. ER Catcher is path-insensitive, therefore it reports false positive UF event races that reside on execution paths that will not execute, e.g., `Service1`. Due to time-insensitivity, ER Catcher reports false positives in cases where the statements execute at specific times, e.g., `SingleActivity4`. ER Catcher fails to filter one false-positive event race in `SingleActivity1` due to incomplete reachability information; however, it is removed in "ER Catcher Prioritized". The imprecision in the off-the-shelf points-to analysis used by ER Catcher leads to a false negative in `Service2`.

We also ran ER Catcher and nAdroid on the Curated dataset, consisting of 31 real-world apps with confirmed event races. ER Catcher was able to analyze all of the 31 apps, while nAdroid could only analyze 27 of them. ER Catcher achieved 100% recall, detecting all of the event races in these apps, while nAdroid achieved 88% recall. Since not all of the event races in these apps are known, we are unable to report the recall using this dataset.

4.2 RQ2: Scalability

We analyzed all three datasets using ER Catcher and nAdroid to compare the scalability of these techniques. Table 3 summarizes the results. For the first two datasets (i.e., BenchERoid and Curated datasets), we did not set a timeout. However, due to the large number of apps in the third dataset (F-Droid), we set a timeout of 5 minutes. On average, ER Catcher analyzed each app in the Curated dataset within 231 seconds, while nAdroid required 3, 134 seconds (~ 52 minutes). Overall, ER Catcher finished the analysis between 12 to 13 times faster than nAdroid for the first two datasets.

For the F-Droid dataset, ER Catcher analyzed 459 out of 500 apps (more than 90%) within the designated time of 5 minutes (two apps could not be analyzed because FlowDroid could not generate the call graph for them). However, nAdroid could only analyze 30 apps (6%) within the allotted time. Furthermore, nAdroid crashed during analysis of 316 apps (more than 60%) indicating nAdroid is unable to complete the analysis irrespective of time.

Table 3: Analysis time summary

		#Analyzed Apps	Average	Median	Min	Max
BenchERoid dataset	nAdroid	29	217s	145s	131s	372s
	ER Catcher	29	18s	20s	7s	35s
	$\frac{\text{nAdroid}}{\text{ER Catcher}}$	1x	12x	7x	18x	10x
Curated dataset	nAdroid	27	3134s	786s	83s	22690s
	ER Catcher	31	231s	29s	4s	5548s
	$\frac{\text{nAdroid}}{\text{ER Catcher}}$	0.87x	13x	27x	21x	4x
F-Droid dataset	nAdroid	30	153s	152s	108s	285*s
	ER Catcher	459	53s	29s	4s	296*s
	$\frac{\text{nAdroid}}{\text{ER Catcher}}$	0.06x	3x	5x	27x	0.96x

4.3 RQ3: Effectiveness of Filters

To evaluate the degree to which ER Catcher filters out false event races in real-world apps, the number of UF event races in three stages of analysis are reported in Table 4. The first row shows the number of event-race candidates identified by the "Event-Race Candidate Analysis" component (recall Section 3.2). The second

row shows the number of filtered UF event races reported by the "Thread-Sensitive Filter" component (recall Section 3.5). The fourth row reports the number of prioritized UF event races (recall Section 3.6). The reduction rates achieved for the Curated and F-Droid datasets are %77 and %86, respectively. This results in prioritizing 37 and 23 UF event races that require manual investigation by developers. In practice, many of the detected event races are caused by the same defect in code, i.e., one defect in code causes event race conditions under multiple execution contexts. As a result, developers can often confirm the presence of a defect without having to review the complete list of reported event races.

Table 4: Effectiveness of filters and prioritization

	Criteria	Average	Median	Min	Max
Curated dataset	#UF Candidates	317	341	0	1238
	#Filtered UFs	150	114	0	540
	% $\frac{\text{Filtered}}{\text{Candidates}}$	50%	54%	0%	92%
	#Prioritized UFs	37	15	0	147
	% $\frac{\text{Prioritized}}{\text{Candidates}}$	23%	12%	0%	100%
F-Droid dataset	#UF Candidates	308	57	0	9249
	#Filtered UFs	129	5	0	1806
	% $\frac{\text{Filtered}}{\text{Candidates}}$	45%	48%	0%	100%
	#Prioritized UFs	23	0	0	1011
	% $\frac{\text{Prioritized}}{\text{Candidates}}$	14%	2%	0%	100%

4.4 RQ4: Impact of Modeling Concurrency

A byproduct of modeling the concurrent behavior of Android apps and libraries in the form of C^2G is that ER Catcher can discover a number of additional methods that are invoked indirectly, i.e., through the library callbacks. This enables ER Catcher to compute reachability of methods in ways that are more accurate than other state-of-the-art techniques, such as FlowDroid [7]. To evaluate this facet of our work, we ran both ER Catcher and FlowDroid on the F-Droid dataset. We identified additional reachable methods in 169 of these apps. Compared to FlowDroid, ER Catcher was able to detect on average 8 more reachable methods per app, and up to 78 more reachable methods in one app.

This is notable given the extensive number of tools (e.g., [8, 9, 23, 48]) that rely on FlowDroid for analysis of Android apps, particularly for security assessment. Consider, for instance, an information leakage vulnerability caused by flow of data from a private source, e.g., camera, to a sink, e.g., network. If the sink is located in a method that is determined to be unreachable by the analysis, the vulnerability cannot be discovered. According to our results, numerous Android security analysis tools [8, 9, 48] built on top of FlowDroid [7] would fail to discover vulnerabilities or malicious behaviors that reside in such locations in code.

5 RELATED WORK

Our work builds upon three major threads of research: concurrency analysis, event-race detection, and summary-based static analysis in Android. We provide a brief overview of the relevant research for each of these threads in this section.

Analysis of Concurrent Behaviors: Researchers have investigated concurrent behaviour of traditional programming languages such as C or Java, primarily focusing on data races [27, 33, 36], deadlocks [11, 14, 31], happens-before relation [52], sequential or non-sequential concurrency errors [35, 49, 50] and testing [42]. None of these, however, has investigated the concurrent behavior of Android apps. The most relevant work is that of Zhou et al. [52] that introduces the concept of Static Vector Clock to analyze *may-happen-in-parallel* relation between instructions in C/C++; however, their realization of this concept is not applicable to event-based systems like Android. ER Catcher addresses this limitation by introducing event-based properties, e.g., *Same Task-Queue Order*.

Researchers have also analyzed the concurrent behavior of Android apps, mostly focusing on dynamic analyses such as test generation [41], manipulating the code execution [16, 44], and detecting happens-before relations [28]. None of these techniques is able to detect event races in Android.

Event Race Detection in Android: Although Android has traditional Java thread constructs, tools developed for Java cannot readily detect event races in Android since existing tools are unaware of relations between events. To meet this gap, researchers have devised dynamic and static event-race detection approaches for Android. Dynamic event race detection approaches gather execution traces of Android apps either manually [21] or automatically [10, 20, 29] using an app crawler such as Monkey [5]. Then the execution traces are analyzed off-line to detect happens-before relations between events either by graph analysis [10, 21, 29], or leveraging vector clock [20]. These techniques fail to identify event races due to their limited coverage of the behaviors of apps [22].

Static event-race detection techniques address the problem of missing true defects by analyzing the whole program. DeVA [38] detects "event anomalies" where two events access the same memory location, and one of the accesses is a write. Since DeVA does not consider happens-before relations, it produces a large number of false positives. SIERRA [22] considers happens-before relation and has limited context-sensitivity (capturing only one asynchronous *action* as the context), but it is not aware of other threads except for the main thread which may result in failure to identify bugs; we addressed this issue by our thread-sensitive analysis modeled in C^3G. nAdroid [17] uses an existing traditional race detection technique to identify Use-after-Free (UF) bugs. Due to the imprecise threadification model of nAdroid it reports a large number of false positives; we extensively explained and empirically evaluated the limitations of nAdroid in comparison to ER Catcher earlier. SARD [45], similar to nAdroid, detects UF bugs, using a flow- and context-sensitive model of Looper. SARD applies an exhaustive context creation strategy making it unscalable. We address the scalability issue by using summary-based analysis techniques to reduce the space complexity.

In all of the mentioned techniques, the support for various Android concurrency libraries is hardcoded in the implementation of tools, making it difficult to revise the tools to support new or modified Android libraries. ER Catcher addresses this limitation by separating the implementation of tool from the library *CSF*s representing the concurrent behavior of Android libraries. Provided with

new or modified library *CSF*s, ER Catcher can be readily extended to support new or modified Android libraries without requiring changes to its implementation. *Summary-Based Static Analysis in Android:* Another research thrust has investigated summary-based approaches for precise and fast analysis [4, 6, 7, 12, 13, 47]. These techniques mostly use Inter-procedural Distributed Environment (IDE) framework [37] for modelling the inter-procedural data-flow of the code; however, to the best of our knowledge, no prior work has leveraged IDE frameworks to model concurrent behavior of code like happens-before relations. Moreover, none of these techniques capture inter-thread communication precisely. As a future work, it would be interesting to see if IDE frameworks, or non-distributive summary based analyses such as [32], can improve the scalability of creating the C^3G model. ER Catcher takes a step towards addressing this issue by introducing ways of summarizing concurrency behavior in Android.

6 CONCLUSION

Concurrency-induced defects, such as event race, are one of the most frequently encountered types of defect in Android apps [51]. We presented ER Catcher, a fast, novel and accurate static analysis framework for event-race detection in Android. Experiments using benchmark apps show that ER Catcher is accurate, capable of detecting event races with 80% precision and 93% recall. Compared to the only other publicly available tool for event-race detection in Android, ER Catcher is substantially faster (by a factor of 12) and more accurate (11% higher F1-measure). Results further corroborate its effectiveness in detecting all of the event races confirmed to exist in a set of 31 real-world apps.

In our future work, we plan to expand the applications of ER Catcher to other concurrency-related analyses. We believe the efficient happens-before analysis of ER Catcher can be used as a plugin in Android Studio to provide real-time feedback to developers during the development. Furthermore, since concurrency is the main cause of flaky tests [43], we aim to study the application of ER Catcher in the detection of such tests.

ER Catcher and research artifacts are publicly available for download from the companion website [2].

ACKNOWLEDGMENT

This work was supported in part by award numbers 1618132 and 1823262 from the National Science Foundation. We would like to thank the anonymous reviewers of this paper for their detailed feedback, which helped us improve the work.

REFERENCES

[1] 2020. *ER Catcher Formal Proofs.* Retrieved August 31, 2020 from https://github.com/seal-hub/ERCatcher/blob/master/ERCatcher_Appendix.pdf
[2] 2020. *ER Catcher Tool.* Retrieved August 31, 2020 from https://github.com/seal-hub/ERCatcher
[3] 2020. *F-Droid.* Retrieved March 4 , 2020 from https://f-droid.org/en/
[4] Karim Ali and Ondřej Lhoták. 2013. Averroes: Whole-program analysis without the whole program. In *European Conference on Object-Oriented Programming*. Springer, 378–400.
[5] Android. 2020. *UI/Application Exerciser Monkey.* Retrieved February 2, 2020 from https://developer.android.com/studio/test/monkey
[6] Steven Arzt and Eric Bodden. 2016. StubDroid: automatic inference of precise data-flow summaries for the android framework. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 725–735.
[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.
[8] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 426–436.
[9] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. 2015. Covert: Compositional analysis of android inter-app permission leakage. *IEEE transactions on Software Engineering* 41, 9 (2015), 866–886.
[10] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable race detection for Android applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015*. ACM Press, Pittsburgh, PA, USA, 332–348. https://doi.org/10.1145/2814270.2814303
[11] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '02)*. Association for Computing Machinery, Seattle, Washington, USA, 211–230. https://doi.org/10.1145/582419.582440
[12] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework.. In *NDSS*.
[13] Tom Deering, Ganesh Ram Santhanam, and Suresh Kothari. 2015. Flowminer: Automatic summarization of library data-flow for malware analysis. In *International Conference on Information Systems Security*. Springer, 171–191.
[14] Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review* 37, 5 (Oct. 2003), 237–252. https://doi.org/10.1145/1165389.945468
[15] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. [n.d.]. Effective Data-Race Detection for the Kernel. ([n. d.]), 12.
[16] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently manifesting asynchronous programming errors in android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 486–497.
[17] Xinwei Fu, Dongyoon Lee, and Changhee Jung. 2018. nAdroid: statically detecting ordering violations in Android applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018*. ACM Press, Vienna, Austria, 62–74. https://doi.org/10.1145/3168829
[18] Sumit Gulwani and Ashish Tiwari. 2007. Computing procedure summaries for interprocedural analysis. In *European Symposium on Programming*. Springer, 253–267.
[19] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.
[20] Chun-Hung Hsiao, Satish Narayanasamy, Essam Muhammad Idris Khan, Cristiano L Pereira, and Gilles A Pokam. 2017. Asynclock: Scalable inference of asynchronous event causality. *ACM SIGPLAN Notices* 52, 4 (2017), 193–205.
[21] Chun-Hung Hsiao, Cristiano L. Pereira, Jie Yu, Gilles A. Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. 2013. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*. ACM Press, Edinburgh, United Kingdom, 326–336. https://doi.org/10.1145/2594291.2594330
[22] Yongjian Hu and Iulian Neamtiu. 2018. Static Detection of Event-based Races in Android Apps. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 257–270. https://doi.org/10.1145/3173162.3173173 event-place: Williamsburg, VA, USA.
[23] R. Jabbarvand, J. Lin, and S. Malek. 2019. Search-Based Energy Testing of Android. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1119–1130.
[24] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.
[25] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using S park. In *International Conference on Compiler Construction*. Springer, 153–169.
[26] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. 2020. Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.
[27] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static analysis of event-driven Node. js JavaScript applications. *ACM SIGPLAN Notices* 50, 10 (2015), 505–519.
[28] Pallavi Maiya and Aditya Kanade. 2017. Efficient computation of happens-before relation for event-driven programs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 102–112.
[29] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for Android applications. *ACM SIGPLAN Notices* 49, 6 (June 2014), 316–325. https:

//doi.org/10.1145/2666356.2594311

[30] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 308–319.

[31] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective Static Deadlock Detection. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 386–396. https://doi.org/10.1109/ICSE.2009.5070538

[32] Rohan Padhye and Uday P Khedker. 2013. Interprocedural data flow analysis in soot using value contexts. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis*. 31–36.

[33] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race detection for web applications. *ACM SIGPLAN Notices* 47, 6 (2012), 251–262.

[34] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 1 (Jan. 2011), 3:1–3:55. https://doi.org/10.1145/1889997.1890000

[35] Shaz Qadeer and Dinghao Wu. 2004. KISS: keep it simple and sequential. *ACM SIGPLAN Notices* 39, 6 (June 2004), 14–24. https://doi.org/10.1145/996893.996845

[36] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 151–166.

[37] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61.

[38] Gholamreza Safi, Arman Shahbazian, William G. J. Halfond, and Nenad Medvidovic. 2015. Detecting event anomalies in event-based systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, Bergamo, Italy, 25–37. https://doi.org/10.1145/2786805.2786836

[39] Navid Salehnamadi, Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. [n.d.]. Poster: A Benchmark for Event-Race Analysis in Android Apps. In *Proceedings of the 18th Annual International Conference on Mobile Systems, Applications, and Services*.

[40] Micha Sharir, Amir Pnueli, et al. 1978. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences ....

[41] Hongyin Tang, Guoquan Wu, Jun Wei, and Hua Zhong. 2016. Generating test cases to expose concurrency bugs in android applications. In *Proceedings of the 31st IEEE/ACM international Conference on Automated software engineering*. 648–653.

[42] Valerio Terragni and Mauro Pezzè. 2018. Effectiveness and challenges in generating concurrent tests for thread-safe classes. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 64–75.

[43] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An empirical study of flaky tests in Android apps. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 534–538.

[44] Jue Wang, Yanyan Jiang, Chang Xu, Qiwei Li, Tianxiao Gu, Jun Ma, Xiaoxing Ma, and Jian Lu. 2018. AATT+: Effectively manifesting concurrency bugs in Android apps. *Science of Computer Programming* 163 (Oct. 2018), 1–18. https://doi.org/10.1016/j.scico.2018.03.008

[45] D. Wu, J. Liu, Y. Sui, S. Chen, and J. Xue. 2019. Precise Static Happens-Before Analysis for Detecting UAF Order Violations in Android. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 276–287. https://doi.org/10.1109/ICST.2019.00035

[46] Xinwei Xie, Jingling Xue, and Jie Zhang. 2013. Acculock: accurate and efficient detection of data races. *Software: Practice and Experience* 43, 5 (2013), 543–576. https://doi.org/10.1002/spe.2121

[47] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2012. Rethinking soot for summary-based whole-program analysis. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. 9–14.

[48] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. 2015. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 303–313.

[49] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. [n.d.]. ConSeq: Detecting Concurrency Bugs through Sequential Errors. ([n. d.]), 14.

[50] Wei Zhang, Chong Sun, and Shan Lu. [n.d.]. ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach. ([n. d.]), 13.

[51] Bo Zhou, Iulian Neamtiu, and Rajiv Gupta. 2015. A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. 1–10.

[52] Qing Zhou, Lian Li, Lei Wang, Jingling Xue, and Xiaobing Feng. 2018. May-happen-in-parallel Analysis with Static Vector Clocks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 228–240. https://doi.org/10.1145/3168813 event-place: Vienna, Austria.